Optimistic Atomic Broadcast: A Pragmatic Viewpoint*

Fernando Pedone André Schiper
Communication Systems Department
Swiss Federal Institute of Technology
CH-1015 – Lausanne – Switzerland

Abstract

This paper presents the Optimistic Atomic Broadcast algorithm (OPT-ABcast) which exploits the *spontaneous total order message reception property* experienced in local area networks in order to allow fast delivery of messages. The OPT-ABcast algorithm is based on a sequence of stages, and messages can be delivered during a stage or at the end of a stage. During a stage, processes deliver messages *fast*. Whenever the spontaneous total order message reception property does not hold, processes terminate the current stage and start a new one by solving a Consensus problem leading to the delivery of some messages. We evaluate the efficiency of the OPT-ABcast algorithms using the notion of deliver latency.

Keywords: optimistic algorithms, atomic broadcast, efficient algorithms, consensus, asynchronous systems

1 Introduction

Atomic Broadcast is a useful abstraction for the development of fault tolerant distributed applications. Understanding the conditions under which Atomic Broadcast is solvable is an important theoretical issue that has been investigated extensively. Solving Atomic Broadcast efficiently is

*A preliminary version of this paper appeared in *Proceedings of the 12th International Symposium on Distributed Computing* (*DISC'98*, pp. 318-322).

also an important and highly relevant pragmatic issue. We present in this paper the *Optimistic Atomic Broadcast* algorithm (called hereafter OPT-ABcast), which allows processes, in certain cases, to deliver messages *fast*. The idea of our OPT-ABcast algorithm stems from the observation that, with high probability, messages broadcast in a local area network are received totally ordered. We call this property *spontaneous total order message reception*.[1] Our algorithm exploits this observation: whenever the spontaneous total order message reception property holds, the OPT-ABcast algorithm delivers messages fast.

The OPT-ABcast algorithm is based on the reduction of Atomic Broadcast to Consensus proposed in [4]. However, contrary to [4], in the OPT-ABcast algorithm, Consensus is not always necessary to deliver messages. Processes executing the OPT-ABcast algorithm see the system evolve as a sequence of stages, and Consensus is only necessary when processes move from one stage to the next. For any stage k, messages can be delivered by some process p, either (1) *during stage k* (i.e., before p executes Consensus), or (2) *at end of stage k* (i.e., after p terminates the k-th Consensus execution). Messages can be delivered much quickly during a stage than at the end of a stage, since the former does not have the cost of executing a Consensus. We evaluate the *efficiency* of the OPT-ABcast algorithm using the notion of *deliver latency*. The efficiency of the OPT-ABcast algorithm is directly related to the spontaneous total order message reception property: the event that triggers the termination of a stage is the violation of this property.

The rest of the paper is structured as follows. Section 2 describes related work, and Section 3 is devoted to the system model and to the definition of deliver latency. In Section 4 we present an overview of the results. Section 5 describes the OPT-ABcast algorithm, and Section 6 discusses its efficiency. Failure handling is discussed in Section 7, and Section 8 concludes the paper.

2 Related Work

The paper is at the intersection of two domains: (1) Atomic Broadcast algorithms, and (2) optimistic algorithms.

The literature on Atomic Broadcast algorithms is abundant (e.g., [1], [3], [4], [5], [7], [10], [13], [15]). However, the multitude of different models (synchronous, asynchronous, etc.) and as-

[1] Spontaneous total order message reception occurs for example with very high probability when network broadcast or IP-multicast are used.

sumptions needed to prove the correctness of the algorithms renders any fair comparison difficult. We base our solution on the Atomic Broadcast algorithm as presented in [4] because it provides a theoretical framework that permits to develop the correctness proofs under assumptions that are realistic in many real systems (i.e., unreliable failure detectors).

Optimistic algorithms have been widely studied in transaction concurrency control (e.g., [2], [11]). To our knowledge, there has been no attempt, prior to this paper, to introduce optimism in the context of agreement algorithms. The closest to the idea presented in the paper is [8], where the authors reduce the Atomic Commitment problem to Consensus and, in order to have a fast decision, exploit the following property of the Consensus problem: if every process starts Consensus with the same value v, then the decision is v. This paper presents a more general idea, and does not require that all the initial values be equal. Moreover, we have here the trade-off of typical optimistic algorithms: if the optimistic assumption is met, there is a benefit (in efficiency), but if the optimistic assumption is not met, there is a loss (in efficiency).

3 System Model and Definitions

3.1 System Model

We consider an asynchronous system composed of n processes $\Pi = \{p_1, \ldots, p_n\}$. A process can only fail by crashing (i.e., we do not consider Byzantine failures). A process that never crashes is *correct*, otherwise it is *faulty*. Processes communicate by message passing, and are connected through FIFO reliable channels, defined by the two primitives $send(m)$ and $receive(m)$. Message m is taken from a set $\mathcal{M}$ to which all messages belong. FIFO reliable channels have the following guarantees: (i) if process q receives message m from p, then p sends m to q *(no creation)*, (ii) if q receives m from p at most once *(no duplication)*, (iii) if p sends m to q, and p and q are correct, then q eventually receives m *(no loss)*, and (iv) if p sends m to q before sending m' to q, then q does not receive m' before receiving m *(FIFO order)*.

Each process p has access to a local failure detector module $\mathcal{D}_p$ that provides (possibly incorrect) information about the processes that are suspected to have crashed. A failure detector may make mistakes, that is, it may suspect a process that has not failed or never suspect a process that has failed. Failure detectors have been classified according to *accuracy* and *completeness* properties which characterise the mistakes they can make [4]. In this paper, we require *(strong*

completeness), that is, eventually every process that crashes is permanently suspected by every correct process. The results presented in the paper are independent of the accuracy property of $\mathcal{D}_p$.

An algorithm A is a collection of n deterministic automata, one per process, and computation proceeds in steps of A. In each step, a process can (1) receive a message that was sent to it, (2) query its failure detector module, (3) modify its state, and (4) send a message to a single process [4]. Informally, a run R of A defines a (possibly infinite) sequence of steps of A, and a problem P is specified by a set of properties that runs must satisfy.

3.2 Consensus

Consensus is defined by the primitives *propose*(v), and *decide*(v), that satisfy the following properties: (i) every correct process eventually decides some value *(termination)*, (ii) every process decides at most once *(uniform integrity)*, (iii) no two correct processes decide differently *(agreement)*, and (iv) if a process decides v, then v was proposed by some process *(uniform validity)*.

Although Consensus is not solvable in asynchronous systems [6], several algorithms are known that solve Consensus in asynchronous systems augmented with failure detectors (e.g., [4], [14]). We do not address this issue in the paper, and assume the existence of an algorithm that solves Consensus.

3.3 Reliable Broadcast and Atomic Broadcast

We assume the existence of a *Reliable Broadcast* primitive, defined by *R-broadcast*(m) and *R-deliver*(m). Reliable Broadcast satisfies the following properties [9]: (i) if a correct process R-broadcasts a message m, then it eventually R-delivers m *(validity)*, (ii) if a correct process R-delivers a message m, then all correct processes eventually R-deliver m *(agreement)*, and (iii) for every message m, every process R-delivers m at most once, and only if m was previously R-broadcast by *sender*(m) *(uniform integrity)*. We assume that the execution of *R-broadcast*(m) results in the execution of *send*(m) to every process p.

Atomic Broadcast is defined by *A-broadcast*(m) and *A-deliver*(m). In addition to the properties of Reliable Broadcast, Atomic Broadcast satisfies the *total order* property [4]: (iv) if two

correct processes p and q A-deliver two messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.

3.4 Deliver Latency

In the following, we introduce the deliver latency as a measure of the efficiency of algorithms solving any Broadcast problem (defined by the primitives α-Broadcast and α-Deliver). The deliver latency is a variation of the Latency Degree introduced [14], which is based on modified Lamport's clocks [12]:

- a *send* event and a *local* event on a process p do not modify p's local clock,
- let $ts(send(m))$ be the timestamp of the $send(m)$ event, and $ts(m)$ the timestamp carried by message m: $ts(m) \stackrel{\text{def}}{=} ts(send(m)) + 1$,
- the timestamp of $receive(m)$ on a process p is the maximum between $ts(m)$ and p's current clock value.

The deliver latency of a message m α-Broadcast in run R of an algorithm A solving a Broadcast problem, denoted $dl^R(m)$, is defined as the difference between (1) the largest timestamp of all α-Deliver(m) events (at most one per process) in run R and (2) the timestamp of the α-Broadcast(m) event in run R. Let π_m^R be the set of processes that α-Deliver message m in run R. The deliver latency of m in R is formally defined as

$$dl^R(m) \stackrel{\text{def}}{=} \mathbf{max}_{p \in \pi_m^R} \left(ts(\alpha\text{-Deliver}_p(m)) - ts(\alpha\text{-Broadcast}(m)) \right).$$

For example, consider a broadcast algorithm A_b where a process p, willing to broadcast a message m, sends m to all processes, each process q on receiving m sends an acknowledge message $ACK(m)$ to all processes, and as soon as q receives n $ACK(m)$ messages, q delivers m. Let R be a run of A_b, as shown in Figure 1, where m is the only message broadcast in R. In this case, $dl^R(m) = 2$.

The deliver latency is a measure of the synchronisation required among processes in a certain run produced by some broadcast algorithm A to deliver a message. With certain care, the deliver latency can be used to characterise the *minimal synchronisation* required among processes by

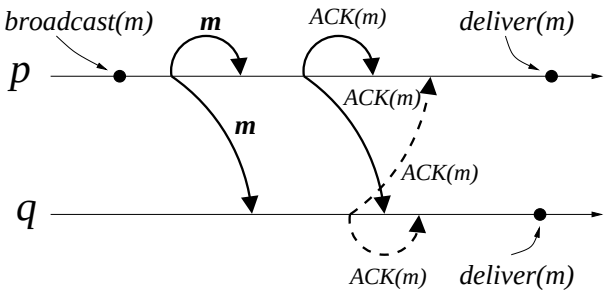

Figure 1: $dl^R(m) = 2$

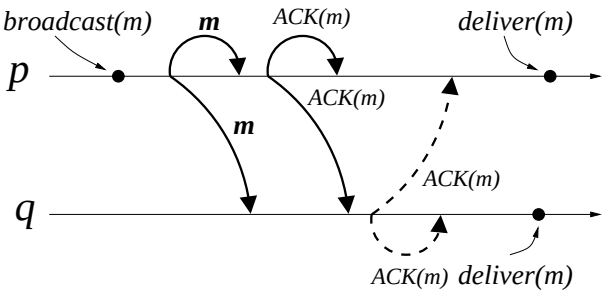

Figure 2: $dl^{R'}(m) = 3$

A to deliver messages. For example, algorithm A_b requires that processes send an $ACK(m)$ message only after receiving message m, and so, no run generated by A_b where m is broadcast will have $send_p(ACK(m))$ preceding $receive_p(m)$, for all process p. Nevertheless, algorithm A_b allows a process q to send $ACK(m)$ after having received $ACK(m)$ from some process p. Thus, there exists a run R' of A_b where m is the only message broadcast, and $receive_q(ACK(m))$ precedes $send_q(ACK(m))$ (see Figure 2). This leads to $dl^{R'}(m) = 3$.

Therefore, when characterising a broadcast algorithm A with the deliver latency parameter, we will consider only the set of runs $\mathcal{R}$ produced by A that exhibit the minimal synchronisation necessary to deliver messages.

4 Overview of the Results

4.1 OPT-ABcast Algorithm

The OPT-ABcast algorithm exploits the spontaneous total order message reception property: if a process p sends a message m to all processes, and a process q sends a message m' to all processes, then the two messages might be received in the same order by all processes. This property typically holds with high probability in local area networks under normal execution conditions (e.g., moderate load). However, under abnormal execution conditions (e.g., high network loads), this property might be violated. More generally, one can consider that the system passes through periods when the spontaneous total order message reception property holds, and periods when the property does not hold.

In the OPT-ABcast algorithm, processes progress in a sequence of stages. Messages can be delivered during a stage or at the end of a stage, and the key aspect is that during a stage,

messages can be delivered faster than at the end of a stage. In order for a process p to deliver messages during a stage k, p has to determine whether the spontaneous total order message reception property holds. Process p determines whether this property holds by exchanging information about the order in which messages are received (see Figure 3). Once p receives this order information from all the other processes, p uses a *prefix* function to determine whether there is a non-empty common sequence of messages received by all processes.

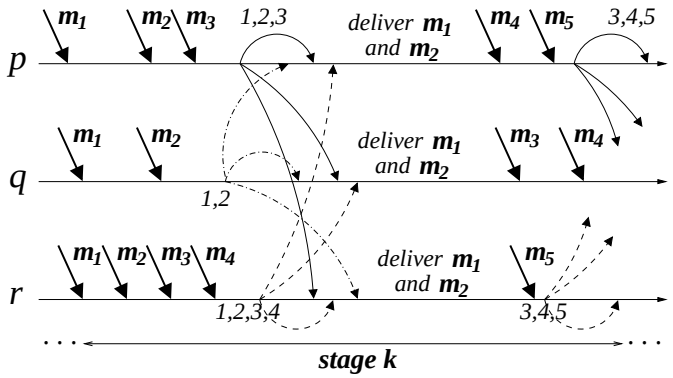

Figure 3: Overview of the OPT-ABcast algorithm (stage k)

Whenever the spontaneous total order message reception property does not hold, processes terminate the current stage, and start a new one (see Figure 4). The termination of a stage involves the execution of a Consensus, which can lead to the deliver of messages. Process failures are discussed in Section 7.

4.2 Deliver Latency of the OPT-ABcast Algorithm

The notion of efficiency is captured by the deliver latency parameter defined in Section 3.4, which informally defines a measure of the synchronisation needed by the OPT-ABcast algorithm to deliver messages. We show that messages delivered during a stage have a deliver latency equal to 2, and messages delivered at the end of a stage have a deliver latency equal to 4. The additional cost *payed* by messages delivered at the end of a stage comes from the Consensus execution. The OPT-ABcast algorithm is based on a Reliable Broadcast and a Consensus, and thus, in order to determine the deliver latency of messages, we use the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14].

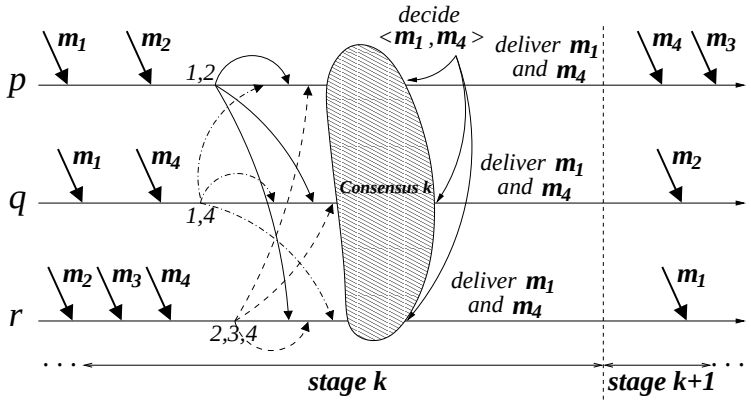

Figure 4: Overview of the OPT-ABcast algorithm (stages k and $k+1$)

Known Atomic Broadcast implementations for the asynchronous model augmented with failure detectors deliver messages with a deliver latency equal to 3. This means that if the spontaneous total order message reception property is violated too frequently, the OPT-ABcast algorithm may become inefficient. However, in case the spontaneous total order message reception property holds frequently, messages can be delivered efficiently using the OPT-ABcast algorithm.

5 The Optimistic Atomic Broadcast Algorithm

5.1 Additional Notation

The Optimistic Atomic Broadcast algorithm presented in the next section handles sequences of messages. In the following we define some terminology needed for the presentation of the algorithm.

A sequence s of messages is denoted by $seq = \langle m_1, m_2, \dots \rangle$. We define the operators $\oplus$ and $\ominus$ for concatenation and decomposition of sequences. Let seq_i and seq_j be two sequences of messages. Then, $seq_i \oplus seq_j$ is the sequence of all the messages in seq_i followed by the sequence of all the messages in seq_j, and $seq_i \ominus seq_j$ is the sequence of all the messages in seq_i that are not in seq_j. The prefix function $\odot$ applied to a set of sequences returns the longest common sequence that is a prefix of all the sequences, or the empty sequence denoted by ϵ.

For example, if $seq_i = \langle m_1, m_2, m_3 \rangle$ and $seq_j = \langle m_1, m_2, m_4 \rangle$, then $seq_i \oplus seq_j = \langle m_1, m_2,$

$m_3, m_1, m_2, m_4\rangle$, $seq_i \ominus seq_j = \langle m_3\rangle$, and $\odot(seq_i, seq_j) = \langle m_1, m_2\rangle$.

5.2 Overview of the OPT-ABcast Algorithm

Algorithm 1 (see page 11) solves Atomic Broadcast. Processes executing Algorithm 1 progress in a sequence of local stages numbered $1, ..., k, ...$ Messages A-delivered by a process *during stage* k are included in the sequence $stgA\_deliver^k$. These messages are A-delivered without the cost of Consensus. Messages A-delivered by a process *at the end of stage* k are included in the sequence $endA\_deliver^k$. These messages are A-delivered with the cost of a Consensus execution. We say that a message m is A-delivered *in stage* k if m is A-delivered either during stage k or at the end of stage k.

Every stage k is terminated by a Consensus to decide on a sequence of messages, denoted by $msgStg^k$. The sequence $msgStg^k$ contains all message that are A-delivered by every process that reaches the end of stage k. Process p starts stage $k+1$ once it has A-delivered all messages in $endA\_deliver^k$ ($endA\_deliver^k = msgStg^k \ominus stgA\_deliver^k$).

The correctness of Algorithm 1 is based on two properties:

1. for any correct processes p and q, all the messages A-delivered by p in stage k are also A-delivered by q in stage k (i.e., $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$), and

2. every sequence of messages A-delivered by some process p in stage k before p executes Consensus k is a non-empty prefix of the sequence decided in Consensus k (i.e., $stgA\_deliver_p^k$ is a prefix of $msgStg^k$).

5.3 Detailed OPT-ABcast Algorithm

All tasks in Algorithm 1 execute concurrently. At each process p, tasks *GatherMsgs* (lines 11-12) and *TerminateStage* (lines 25-35) are started at initialisation time. Task $StgDeliver^k$ (lines 13-24) is started by p when p begins stage k. Lines 20 and 21 in task $StgDeliver^k$ are atomic, that is, task $StgDeliver^k$ is not interrupted (by task *TerminateStage*) after it has executed line 20 and before having executed line 21. Process p in stage k manages the following sequences.

- $R\_delivered_p$: contains all messages R-delivered by p up to the current time,

- $A\_delivered_p$: contains all messages A-delivered by p up to the current time,
- $stgA\_deliver_p^k$: is the sequence of messages A-delivered by p during stage k, up to the current time,
- $endA\_deliver_p^k$: is the sequence of messages A-delivered by p at the end of stage k.

When p wants to A-broadcast message m, p executes *R-broadcast*(m) (line 9). After p R-delivers a message m (line 11), p includes m in $R\_delivered_p$, and eventually executes task $StgDeliver^k$ (line 13). At task $StgDeliver^k$, p sends a sequence of messages that it has not A-delivered yet to all processes (line 14), and waits for such sequence from all processes (line 15). The next actions executed by p depend on the messages it receives at the *wait* statement (line 15).

1. If p receives a sequence from all processes, and there is a non-empty prefix common to all these sequences, p A-delivers the messages in the common prefix (line 20). If not, p R-broadcasts message (k, ENDSTG) to terminate the current stage k (line 23).
2. Once p R-delivers message (k, ENDSTG) at line 25, p terminates task $StgDeliver^k$ (line 26), and starts the k-th Consensus execution (line 27), proposing a sequence of all messages p has R-delivered up to the current time but not A-delivered in any stage k', $k' < k$. Upon deciding for Consensus k (line 28), p builds the sequence $endA\_deliver^k$ (line 29) and A-delivers the messages in $endA\_deliver^k$ (line 30). Process p then starts stage $k+1$ (lines 32-35).

5.4 Proof of Correctness

The correctness of the OPT-ABcast algorithm follows from Propositions 5.1 (Agreement), 5.2 (Partial Order), 5.3 (Validity), and 5.4 (Integrity). In order to prove some results that follow, we consider the number of times that processes execute lines 13-21 in a given stage. Hereafter, $stgA\_deliver_p^{k,l_k}$ denotes the value of $stgA\_deliver_p^k$ after p executes line 21 for the l_k-th time in stage k, $l_k > 0$, and $stgA\_deliver_p^{k,0}$ denotes the value of $stgA\_deliver_p^k$ before p executes lines 13-21 for the first time (i.e., $stgA\_deliver_p^{k,0} = \epsilon$).

Lemma 5.1 *If p and q are two processes that execute the l_k-th iteration of line 21 in stage k, then $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$.*

Algorithm 1 OPT-ABcast algorithm

1: Initialisation (see Section 5.3 for a description of the variables):
2: $R\_delivered \leftarrow \epsilon$
3: $A\_delivered \leftarrow \epsilon$
4: $k \leftarrow 1$
5: $stgA\_deliver^k \leftarrow \epsilon$
6: $endA\_deliver^k \leftarrow \epsilon$
7: **fork tasks** { *GatherMsgs*, $StgDeliver^1$, *TerminateStage* }

8: To execute *A-broadcast*(m):

9: *R-broadcast*(m)

10: *A-deliver*$(-)$ occurs as follows:

11: **when** *R-deliver*(m) {***Task*** *GatherMsgs*}
12: $R\_delivered \leftarrow R\_delivered \oplus \langle m \rangle$

13: **when** $(R\_delivered \ominus A\_delivered) \ominus stgA\_deliver^k \neq \epsilon$ {***Task*** $StgDeliver^k$}
14: send $(k, (R\_delivered \ominus A\_delivered) \ominus stgA\_deliver^k)$ to all
15: **wait until** for $[\forall q \in \Pi :$ received $(\underline{k}, msgSeq_q)$ from q **or** $\mathcal{D}_p \neq \emptyset]$
16: $\pi = \{ q \mid p$ received $(k, msgSeq_q)$ from $q \}$
17: $prefix \leftarrow \odot_{\forall q \in \pi} msgSeq_q$
18: **if** $\pi = \Pi$ **and** $prefix \neq \epsilon$ **then**
19: $stgDeliver \leftarrow prefix \ominus stgA\_deliver^k$
20: [deliver all messages in $stgDeliver$ following their order in $stgDeliver$;
21: $stgA\_deliver^k \leftarrow stgA\_deliver^k \oplus prefix$]
22: **else**
23: *R-broadcast*(k, ENDSTG)
24: **end task**

25: **when** *R-deliver*$(\underline{k}, \underline{\text{ENDSTG}})$ {***Task*** *TerminateStage*}
26: **terminate task** $StgDeliver^k$, if executing
27: *propose*$(k, R\_delivered \ominus A\_delivered)$
28: **wait until** *decide*$(\underline{k}, msgStg^k)$
29: $endA\_deliver^k \leftarrow msgStg^k \ominus stgA\_deliver^k$
30: deliver all messages in $endA\_deliver^k$ following their order in $endA\_deliver^k$
31: $A\_delivered \leftarrow A\_delivered \oplus (stgA\_deliver^k \oplus endA\_deliver^k)$
32: $k \leftarrow k + 1$
33: $stgA\_deliver^k \leftarrow \epsilon$
34: $endA\_deliver^k \leftarrow \epsilon$
35: **fork task** $StgDeliver^k$

PROOF. We first show that for any l, $0 < l \leq l_k$, $prefix_p^l = prefix_q^l$. Since p and q execute line 21 for the l-th time in stage k, p and q receive a message of the type $(k, msgSeq)$ from every process in the l-th iteration of lines 15. From line 17 and the fact that communication between processes follows a FIFO order, $prefix_p^l = \odot_{\forall r} msgSeq_r^l$, and $prefix_q^l = \odot_{\forall r} msgSeq_r^l$, where $msgSeq_r^l$ is the l-th message of the type $(k, msgSeq_r)$ received from process r, and we conclude that $prefix_p^l = prefix_q^l$. From line 21, $stgA\_deliver^{k,l} = stgA\_deliver^{k,l-1} \oplus prefix^l$, and a simple induction on l_k leads to $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$. □

Lemma 5.2 *For any process p, and all $k \geq 1$, if p executes $decide(k, msgStg^k)$, then $stgA\_deliver_p^k$ is a prefix of $msgStg^k$.*

PROOF. Assume that p executes $decide(k, msgStg^k)$. By uniform validity of Consensus, there is a process q that executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$. Let l_k be the number of times that p executes line 21 before executing $decide(k, -)$. We show by induction on l_k that $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. BASIC STEP. ($l_k = 0$) In this case, $stgA\_deliver_p^{k,0} = \epsilon$ and the lemma is trivially true. INDUCTIVE STEP. Assume that the lemma holds for $l'_k - 1$, $0 < l'_k < l_k$. We show that $stgA\_deliver_p^{k,l'_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. From line 21, $stgA\_deliver_p^{k,l'_k} = stgA\_deliver_p^{k,l'_k-1} \oplus prefix_p^{l'_k}$. Since communication follows FIFO order, $prefix_p^{l'_k} = \odot_{\forall q}((R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{l'_k-1})$. From Lemma 5.1, $prefix_p^{l'_k} = \odot_{\forall q}(R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_p^{l'_k-1}$. Thus, $stgA\_deliver_p^{k,l'_k} = stgA\_deliver_p^{k,l'_k-1} \oplus (\odot_{\forall q}(R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_p^{l'_k-1})$. Since by the induction hypothesis, $stgA\_deliver_p^{l'_k-1}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$, $stgA\_deliver_p^{k,l'_k} = \odot_{\forall q}(R\_delivered_q \ominus A\_delivered_q)$, and we conclude that $stgA\_deliver_p^{k,l'_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. □

Lemma 5.3 *For any two correct processes p and q, and all $k \geq 1$, if p A-delivers messages in $endA\_deliver_p^k$, then q A-delivers messages in $endA\_deliver_q^k$.*

PROOF. If p A-delivers messages in $endA\_deliver_p^k$, then p executes the $decide(k, msgStg^k)$ statement at line 28, and the $propose(k, -)$ statement at line 27. Therefore, p R-delivers a message of the type (k, ENDSTG) at line 25. By the agreement property of Reliable Broadcast, q eventually R-delivers message (k, ENDSTG), and executes the $propose(k, -)$ statement at line 27. By

agreement of Consensus, q executes the $decide(k, msgStg^k)$ statement, and A-delivers messages in $endA\_deliver_q^k$ at line 30. □

Lemma 5.4 *For any two processes p and q, and all $k \geq 1$, if both p and q execute line 29, then $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$.*

PROOF. From line 29, $endA\_deliver_p^k = msgStg^k \ominus stgA\_deliver_p^k$, and so, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_p^k \oplus (msgStg^k \ominus stgA\_deliver_p^k)$. By Lemma 5.2, $stgA\_deliver_p^k$ is a prefix of $msgStg^k$, and so, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = msgStg^k$. From a similar argument, $stgA\_deliver_q^k \oplus endA\_deliver_q^k = msgStg^k$. Therefore, we conclude that $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. □

Lemma 5.5 *For any process p, and all $k \geq 1$, if message $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$ then there is no k', $k' < k$, such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.*

PROOF. The proof is by contradiction. Assume that there exist a process p, a message m, some k, and some $k' < k$, such that $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$. We distinguish two cases: (a) $m \in stgA\_deliver_p^k$, or (b) $m \in endA\_deliver_p^k$. Note that from line 29, it cannot be that $m \in stgA\_deliver_p^k$ and $m \in endA\_deliver_p^k$.

Case (a). From lines 21, 17 and 15 $stgA\_deliver_p^k$ is a common non-empty prefix among the messages of the type $(k, msgSeq)$ received by p from all processes. Thus p has received the message $(k, msgSeq_p)$ (i.e., a message that p sent to itself), such that $m \in msgSeq_p$. But $msgSeq_p = R\_delivered_p \ominus A\_delivered_p$ (line 14), and so, $m \notin A\_delivered_p$. When p executes line 14 at stage k, $A\_delivered_p = \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$. This follows from line 31, the only line where $A\_delivered$ is updated. Therefore, $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, contradicting the fact that there is a $k' < k$ such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.

Case (b). From line 29, $m \in msgStg^k$, and from line 28, and validity of Consensus, there is a process q that executes $propose(k, R\_delivered_q \ominus A\_delivered_q)$ such that $m \in R\_delivered_q \ominus A\_delivered_q$. So, $m \notin A\_delivered_q$. Since when q executes line 27, $A\_delivered_q = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i), m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$, and from Lemma 5.4 $\oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i) = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$. Thus, $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, a contradiction that concludes the proof. □

Proposition 5.1 (AGREEMENT). *If a correct process p A-delivers a message m, then every correct process q eventually A-delivers m.*

PROOF: Consider that p has A-delivered message m in stage k. We show that q also A-delivers m in stage k. There are two cases to consider: (a) p A-delivers messages in $endA\_deliver_p^k$, and (b) p does not A-deliver messages in $endA\_deliver_p^k$.

Case (a). From Lemma 5.3 and the fact that p A-delivers messages in $endA\_deliver_p^k$, q A-delivers messages in $endA\_deliver_q^k$, and from Lemma 5.4, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Since p A-delivers m in stage k, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and so, $m \in stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Therefore, q either A-delivers m at line 20 (in which case $m \in stgA\_deliver_q^k$), or at line 30 (in which case $m \in stgA\_deliver_q^k$).

Case (b). Since p does not A-deliver messages in $endA\_deliver_p^k$, from Lemma 5.3, no correct process q A-delivers messages in $endA\_deliver_q^k$. However, m is A-delivered in stage k by p, and so, it must be that $m \in stgA\_deliver_p^k$. Assume that $m \in stgA\_deliver_p^{k,l_k}$, where l_k is such that for any $l'_k < l_k$, $m \notin stgA\_deliver_p^{k,l'_k}$. Therefore, p executes the l_k-th iteration of line 21 in stage k, and we claim that q also executes the l_k-th iteration of line 21 in stage k. The claim is proved by contradiction. From the algorithm, q executes *R-broadcast*$(k, -)$. By agreement and validity of Reliable Broadcast, every correct process R-delivers the message (k, ENDSTG) and executes *propose*$(k, -)$. By agreement and termination of Consensus, every correct process decides on Consensus k, and eventually A-delivers messages in $endA\_deliver^k$, contradicting the fact that no correct process A-delivers messages in $endA\_deliver^k$, and concluding the proof of the claim. Since p and q execute the l_k-th iteration of line 21 in stage k, and $m \in stgA\_deliver_p^{k,l'_k}$, from Lemma 5.1, $m \in stgA\_deliver_q^{k,l'_k}$, and from lines 20-21, q A-delivers m. □

Proposition 5.2 (TOTAL ORDER). *If correct processes p and q both A-deliver messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.*

PROOF: Assume that p A-delivers message m in stage k, and m' in stage k', $k' > k$. Therefore, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m' \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$, and it follows immediately from Lemma 5.4 that q A-delivers m before m'. Now, assume that m and m' are A-delivered by p in stage k. Therefore, m precedes m' in $stgA\_deliver_p^k \oplus endA\_deliver_p^k$.

By Lemma 5.4, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$, and so, from Tasks $stgDeliver^k$ (line 20), *TerminateStage* (line 30), q A-delivers m before m'. □

Proposition 5.3 (VALIDITY). *If a correct process p A-broadcasts a message m, then p eventually A-delivers m.*

PROOF: For a contradiction, assume that p A-broadcasts m but never A-delivers it. From Proposition 5.1, no correct process A-delivers m. Since p A-broadcasts m, it R-broadcasts m, and from the validity of Reliable Broadcast, p eventually R-delivers m and includes m in $R\_delivered_p$. Since no correct process A-delivers m, $m \notin A\_delivered_p$, and $m \notin stgA\_deliver^k$. From the agreement of Reliable Broadcast, there is a stage k_1 such that for all $l \geq k_1$, and every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^l$.

Let k_2 be a stage such that for all $l \geq k_2$ every faulty process has crashed (i.e., no faulty process executes stage l), and let $k \geq max(k_1, k_2)$. Thus, no faulty process executes stage k, and for every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^k$ at stage k. Let r be a correct process that executes the *when* statement at line 13. At line 15, r suspects some faulty process and eventually executes $R\text{-}broadcast(k, \text{ENDSTG})$. By the validity and agreement of Reliable Broadcast, every correct process q R-delivers (k, ENDSTG) and executes $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $m \in R\_delivered_q \ominus A\_delivered_q$. By agreement and termination of Consensus, every q decides on the same $msgStg^k$, and by validity of Consensus $m \in msgStg^k$. It follows that q A-delivers m, a contradiction that concludes the proof. □

Proposition 5.4 (UNIFORM INTEGRITY). *For any message m, each process A-delivers m at most once, and only if m was previously A-broadcast by $sender(m)$.*

PROOF: We first show that, for any message m, each process A-delivers m only if m was previously A-broadcast by $sender(m)$. There are two cases to consider. (a) A process p A-delivers m at line 20. Thus, p received a message $(k, msgSeq_q)$ from every process q, for some k, and $m \in msgSeq_q$. From line 14, $m \in R\_delivered_q$, and from line 12, p has R-delivered m. By uniform integrity of Reliable Broadcast, $sender(m)$ R-broadcasts m, and so, $sender(m)$ A-broadcasts m. (b) Process p A-delivers m at line 30. Thus, from line 29, $m \in msgSet^k$, for some k, and p executed $decide(k, msgStg^k)$. By uniform validity of Consensus, some process q

executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $m \in R\_delivered_q \ominus A\_delivered_q$. It follows from an argument similar to the one presented in item (a) that $sender(m)$ A-broadcasts m.

We now show that m is only A-delivered once by p. From Lemma 5.5, it is clear that if m is A-delivered in stage k (i.e., $m \in stgA\_deliver^k \oplus endA\_deliver^k$), then m is not A-delivered in some other stage k', $k' \neq k$. It remains to be proved that m is not A-delivered more than once in stage k. If m is A-delivered at line 20, then $m \in stgA\_deliver^k$ (note that lines 20 and 21 cannot be interrupted by the *terminate task* statement at line 26), and from line 29, if $m \in stgA\_deliver^k$, $m \notin endA\_deliver^k$, and this concludes the proof. □

Theorem 5.1 *Algorithm 1 solves Atomic Broadcast.*

Proof. Immediate from Propositions 5.1, 5.2, 5.3, and 5.4. □

6 Efficiency of the OPT-ABcast Algorithm

6.1 On the Necessity of Consensus

In this section, we discuss the efficiency of the OPT-ABcast algorithm. Intuitively, the idea is that if Consensus is not needed to deliver some message m, but necessary to deliver some other message m', then the deliver latency of m' is greater than the deliver latency of m. Before going into details about the deliver latency of messages delivered with and without the cost of a Consensus execution (see Section 6.2), we present a more general result about the necessity of Consensus in the OPT-ABcast algorithm. Briefly, Proposition 6.1 states that in a failure free and suspicion free run, Consensus is not executed in stage k if the spontaneous total order message reception property holds in k.

Proposition 6.1 *Let R be a failure free and suspicion free run of the OPT-ABcast algorithm. If for every two processes p and q, and all $l_k > 0$, $(R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k} \odot (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k} \neq \epsilon$, then no process executes Consensus k in R.*

Proof. Assume that there is a process p that executes Consensus k in R. From the algorithm, p R-delivers a message of the type (k, EndStg), and by uniform integrity of Reliable Broadcast,

some process r executed *R-broadcast*(k, ENDSTG). From line 18, either (a) q suspects some process, or (b) there is an $l_k \geq 0$, such that $prefix_q^{l_k+1} = \epsilon$. Case (a) contradicts the hypothesis that no process is suspected, so it must be that $prefix_q^{l_k+1} = \epsilon$.

From lines 17, 14 and 15, $prefix_q^{l_k+1} = \odot_{\forall r} msgSeq_r^{l_k+1} = \odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k}$, and so, $\odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k} = \epsilon$. Therefore, there must exist two processes p and q such that $(R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k} \oplus (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k} = \epsilon$, contradicting the hypothesis. □

Thus, from Proposition 6.1, in a failure free and suspicion free run, Consensus is only necessary in stage k if the spontaneous total order message reception property does not hold in k.

6.2 Deliver Latency of the OPT-ABcast Algorithm

We now discuss in more detail the efficiency of the OPT-ABcast algorithm. For every process p and all stages k, there are two cases to consider: (a) messages A-delivered by p during stage k (line 20), and (b) messages A-delivered by p at the end of stage k. The main result is that for case (a), the Optimistic Atomic Broadcast algorithm can A-deliver messages with a deliver latency equal to 2, while for case (b), the deliver latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a deliver latency of at least 3, these results show the tradeoff of the Optimistic Atomic Broadcast algorithm: if the spontaneous total order message reception property only holds rarely, the OPT-ABcast algorithm is not attractive, while otherwise, the OPT-ABcast algorithm leads to smaller costs compared to known Atomic Broadcast algorithms.

Propositions 6.2 and 6.3 assess the minimal cost of the Optimistic Atomic Broadcast algorithm to A-deliver a message m. Proposition 6.2 defines a lower bound on the deliver latency of m, and Proposition 6.3 states that this bound can be reached in runs where no process A-delivers m at the end a of stage. We consider a particular implementation of Reliable Broadcast that appears in [4].[2]

[2]Whenever a process p wants to R-broadcast a message m, p sends m to all processes. Once a process q receives m, if $q \neq p$ then q sends m to all processes, and, in any case, q R-delivers m.

Proposition 6.2 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [4]. If $\mathcal{R}$ is a set of runs generated by Algorithm 1 such that m is a message A-delivered in runs in $\mathcal{R}$, then there is no run R, $R \in \mathcal{R}$, such that $dl^R(m) < 2$.*

PROOF. Assume that m is A-delivered in stage k, and let p be a process that A-delivers m in R. There are two cases to consider: (a) m is A-delivered by p during stage k, and (b) m is A-delivered by p at the end of stage k. In case (a), p received a message $(-, msgSeq_q)$ from every process q such that $m \in msgSet_q$. Since q executes $send(-, R\_delivered_q \ominus A\_delivered_q)$ such that $m \in R\_delivered_q \ominus A\_delivered_q$, q executes *R-deliver*(m), and by uniform integrity of Reliable Broadcast, there is some process r that executes *R-broadcast*(m), which is the process that executes *A-broadcast*(m). From the implementation of Reliable Broadcast, $ts(A\text{-}broadcast_r(m)) = ts(send_r(m))$, and by the definition of delivery latency, $ts(A\text{-}deliver_p(m)) = ts(send_r(m)) + 2$, and so, $dl^R(m) \geq 2$.

In case (b), it follows that p executes *R-deliver*$(-, \text{ENDSTG})$, and so, there is some process q that executes *R-broadcast*$(-, \text{ENDSTG})$ (line 23). Since q executes line 23, it must be that $m \in R\_delivered_q \ominus A\_delivered_q$, and so, q R-delivered m from some r. From an argument similar to the one presented in case (a), $dl^R(m) \geq 2$. □

The proof for Proposition 6.3 has been omitted since it is very similar to the proof of Proposition 6.2.

Proposition 6.3 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [4]. If $\mathcal{R}$ is a set of runs generated by Algorithm 1, such that in runs in $\mathcal{R}$, m is a message only A-delivered during stage k, for some $k > 0$, then there is a run R, $R \in \mathcal{R}$, such that $dl^R(m) = 2$.*

The results that follow define the behaviour of the Optimistic Atomic Broadcast algorithm for messages A-delivered at the end of stage k. Proposition 6.4 establishes a lower bound for this case, and Proposition 6.5 shows that this bound can be reached when there are no process failures and no failure suspicions.

Proposition 6.4 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14]. Let $\mathcal{R}$ be a set of runs generated by Algorithm 1, such that m and m' are the only messages A-broadcast and A-delivered*

in $\mathcal{R}$. If m and m' are A-delivered at line 30 by some process p, then there is no run R, $R \in \mathcal{R}$, such that $dl^R(m) < 4$ and $dl^R(m') < 4$.

PROOF. Assume for a contradiction that there is a run R in $\mathcal{R}$ such that $dl^R(m) < 4$ and $dl^R(m') < 4$. Since p A-delivers m and m' at line 30, p R-delivers message $(-, \text{ENDSTG})$, and by uniform integrity of Reliable Broadcast, there is a process q that executes $R\text{-}broadcast(-, \text{ENDSTG})$. Therefore, q has R-delivered at least one message that is neither in $A\_delivered_q$ nor in $stgA\_deliver_q$ (line 13). Without loss of generality, assume that this message is m. Since q R-delivered m, there is a process r that executes $R\text{-}broadcast(m)$, and this is the process that executes $A\text{-}broadcast(m)$. From the definition of deliver latency, we have that $ts(propose_p(-)) = ts(A\text{-}broadcast_r(m)) + 2$. From the contradiction hypothesis, $dl^R(m) = ts(A\text{-}deliver_p(m)) - ts(A\text{-}broadcast_r(m)) < 4$, and so, $ts(A\text{-}deliver_p(m)) = ts(A\text{-}broadcast_r(m)) + 2 + c < 4$, where c is the length of the causal chain of messages generated by the Consensus execution (i.e., between $propose_p(-)$ and $decide_p(-)$). We conclude that $c < 2$. This leads to a contradiction since for the Consensus algorithm presented in [14], the minimal causal chain of messages is 2, and therefore, $c \geq 2$. □

The proof for Proposition 6.5 has also been omitted since it follows an argument similar to Proposition 6.4.

Proposition 6.5 *Assume that Algorithm 1 uses the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14]. Let $\mathcal{R}$ be a set of runs generated by Algorithm 1, such that in every run in $\mathcal{R}$, m and m' are the only messages A-broadcast and A-delivered, and there are no process failures and no failure suspicions. If m and m' are A-delivered at line 30 by some process p, then there is a run R, $R \in \mathcal{R}$, such that $dl^R(m) = 4$ and $dl^R(m') = 4$.*

7 Handling Failures

In the OPT-ABcast algorithm (line 18), whenever task $StgDeliver^k$ does not receive messages from all processes in Π, the current stage k is terminated, which leads to an execution of Consensus to A-deliver the messages. Therefore, as soon as a process $p \in \Pi$ crashes, the A-deliver of messages will always be *slow* (i.e., with a deliver latency of at least 4). This can easily

be solved by adding a membership service to our OPT-ABcast algorithm as follows. Let v_i be the current view of system Π ($v_i \subseteq \Pi$):

- at line 18, replace condition $\pi = \Pi$ by $\pi = v_i$.

Once a process p crashes (or is suspected to have crashed), p is removed from the view, and *fast* A-deliver of messages is again possible. We do not discuss further this extension to the OPT-ABcast algorithm, but we note that the instance of the membership problem needed to remove a crashed process can easily be integrated into the Consensus problem that terminates a stage.

8 Conclusion

This work originated from the pragmatic observation that, with high probability, messages broadcast in a local area network are "spontaneously" totally ordered. Exploiting this observation led us to develop the Optimistic Atomic Broadcast algorithm. Processes executing the OPT-ABcast algorithm progress in a sequence of stages, and messages can be delivered during stages or at the end of stages. Messages are delivered faster during stages than at the end of stages. For any process, the current stage is terminated, and another one started, whenever the spontaneous total order message reception property does not hold.

The efficiency of the OPT-ABcast algorithm has been quantified using the notion of deliver latency. The deliver latency of messages delivered during a certain stage has been shown to be equal to 2, while the deliver latency of messages delivered at the end of a stage equal to 4. This result shows the tradeoff of the OPT-ABcast algorithm: if most messages are delivered during the stages, the OPT-ABcast algorithm outperforms known atomic broadcast algorithms, otherwise, the OPT-ABcast algorithm is outperformed by known atomic broadcast algorithms.

Finally, to the best of our knowledge, the OPT-ABcast algorithm is the first agreement algorithm to exploit an *optimistic* property. If this property is satisfied the efficiency of the algorithm is improved, if the property is not not satisfied the efficiency of the algorithm deteriorates (however the optimistic property has no impact on the safety and liveness guarantees of the system). We believe that this opens interesting perspectives for revisiting or improving other agreement algorithms.

References

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. Number 13, pages 551–560, May 1993.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[5] J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[6] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[7] H. Garcia-Molina and A. Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[8] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 692–697, May 1996.

[9] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press/Addison-Wesley, second edition, 1993.

[10] P. Jalote. Efficient ordered broadcasting in reliable csma/cd networks. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 112–119, May 1998.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 90.

[14] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[15] U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.